OBJECT ORIENTED PROGRAMMING AND IT CONCEPTS

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of objects. These objects are instances of classes, which can contain data (attributes) and methods (functions). OOP principles include encapsulation, inheritance, polymorphism, and abstraction.

1. ENCAPSULATION

Encapsulation involves bundling the data (attributes) and methods that operate on the data into a single unit, or class, and restricting access to some of the object's components. Encapsulation is a fundamental OOP concept where the data (variables) and methods that operate on the data are bundled together. This allows us to control access and modification through accessors (getters) and mutators (setters).

Example 1:

```
class BankAccount {

    private String accountNumber;

    private double balance;


    public BankAccount(String accountNumber, double balance) {

        this.accountNumber = accountNumber;

        this.balance = balance;

    }


    public String getAccountNumber() {

        return accountNumber;

    }


    public double getBalance() {

        return balance;
```

```
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public boolean withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            return true;
        } else {
            return false;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("123456", 1000.00);

        // Display initial balance
```

```
        System.out.println("Initial Balance: " + account.getBalance());

        // Deposit money
        account.deposit(500.00);
        System.out.println("Balance after deposit: " + account.getBalance());

        // Withdraw money
        boolean success = account.withdraw(200.00);
        if (success) {
            System.out.println("Balance after withdrawal: " + account.getBalance());
        } else {
            System.out.println("Withdrawal failed. Insufficient balance.");
        }
    }
}
```

Key Concepts:

- **Private Variables:** accountNumber and balance are private, so they cannot be accessed directly from outside the class.
- **Public Methods:** getAccountNumber(), getBalance(), deposit(), and withdraw() provide controlled access.

Real-World Example: ATM

Think of an ATM machine:

- **Data Encapsulation:** The ATM interface lets you interact with your account balance and transaction history without showing how these operations are implemented internally.
- **Controlled Access:** You use your card and PIN to access your account details securely. The ATM (analogous to methods) provides a safe way to deposit and withdraw money.

Benefits of Encapsulation

1. **Data Hiding:** Sensitive data is hidden from outside access.
2. **Increased Security:** Only specific methods can change or access data.
3. **Flexibility:** Internal implementation can change without affecting users.
4. **Maintainability:** Easier to manage and update cod

Example 2: Student Class

```
class Student {
    private String name;
    private int age;

    // Constructor
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }
```

```
    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student("Alice", 20);

        // Display initial details
        System.out.println("Name: " + student.getName());
        System.out.println("Age: " + student.getAge());

        // Modify details
```

```
        student.setName("Bob");

        student.setAge(25);


        // Display updated details

        System.out.println("Updated Name: " + student.getName());

        System.out.println("Updated Age: " + student.getAge());

    }

}
```

Real-World Analogy: School Administration

- **Data Protection:** A student's personal information is managed securely by the administration.
- **Controlled Modifications:** Only authorized staff can update student records, ensuring data integrity.

Encapsulation is crucial for creating robust and secure applications by protecting the internal state and ensuring consistent interfaces for interaction.

2. INHERITANCE IN JAVA

Definition: Inheritance is a mechanism where one class (child class) can inherit fields and methods from another class (parent class). It promotes code reusability and establishes a relationship between classes.

Types of Inheritance

1. **Single Inheritance:** A class inherits from one superclass.
2. **Multilevel Inheritance:** A class is derived from another derived class.
3. **Hierarchical Inheritance:** Multiple classes inherit from a single superclass.
4. **Multiple Inheritance (not supported directly in Java):** A class inherits from multiple classes. Java handles this through interfaces.

Java Example: Single Inheritance

```
// Parent class

class Vehicle {
```

```
    public void start() {
        System.out.println("Vehicle starting...");
    }
}

// Child class
class Car extends Vehicle {
    public void honk() {
        System.out.println("Car honking...");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();  // Inherited method
        myCar.honk();   // Own method
    }
}
```

Java Example: Multilevel Inheritance

```
// Base class
class Animal {
    public void eat() {
        System.out.println("Eating...");
    }
}
```

```
// Derived class
class Mammal extends Animal {
    public void walk() {
        System.out.println("Walking...");
    }
}

// Further derived class
class Dog extends Mammal {
    public void bark() {
        System.out.println("Barking...");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();   // Inherited from Animal
        myDog.walk();  // Inherited from Mammal
        myDog.bark();  // Own method
    }
}
```

Example 2: Hierarchical Inheritance

```
// Base class
class Shape {
    public void draw() {
        System.out.println("Drawing shape...");
```

```
    }
}

// Derived class
class Circle extends Shape {
    public void area() {
        System.out.println("Area of circle...");
    }
}

// Another derived class
class Square extends Shape {
    public void area() {
        System.out.println("Area of square...");
    }
}

public class Main {
    public static void main(String[] args) {
        Circle myCircle = new Circle();
        Square mySquare = new Square();
        myCircle.draw();  // Inherited method
        mySquare.draw();  // Inherited method
        myCircle.area();  // Own method
        mySquare.area();  // Own method
    }
}
```

Real-World Examples

1. **Vehicle System:**
    - **Superclass:** Vehicle
    - **Subclass:** Car, Bike, Truck
    - **Common Methods:** start(), stop()
2. **Organization Hierarchy:**
    - **Superclass:** Employee
    - **Subclass:** Manager, Developer
    - **Common Methods:** getSalary(), work()

Key Points

- **extends Keyword:** Used to establish inheritance.
- **Reusability:** Inherited classes reuse existing code, reducing redundancy.
- **Polymorphism:** Inherited classes can override methods to perform specific behaviors.
- **Access Control:** Inherited classes can access protected and public members of the superclass.

Benefits of Inheritance

- **Code Reusability:** Share common code across classes.
- **Hierarchy Representation:** Models real-world relationships.
- **Maintainability:** Easier to update and manage code.

By using inheritance, you can create a well-structured and efficient codebase that reflects real-world relationships and simplifies complex systems.

3. Polymorphism in Java

Definition: Polymorphism allows objects to be treated as instances of their parent class. It enables a single action to behave differently based on the object performing it.

Types of Polymorphism

1. **Compile-time Polymorphism (Method Overloading):**
    - Occurs when multiple methods have the same name but different parameters.
    - Resolved during compile time.
2. **Runtime Polymorphism (Method Overriding):**
    - Occurs when a subclass provides a specific implementation of a method declared in its parent class.
    - Resolved at runtime.

Example 1: Compile-time Polymorphism (Method Overloading)

```
class Printer {
    // Overloaded methods with different parameters
    void print(int num) {
        System.out.println("Printing number: " + num);
    }

    void print(String text) {
        System.out.println("Printing text: " + text);
    }
}

public class Main {
    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.print(5);       // Calls print(int num)
        printer.print("Hello"); // Calls print(String text)
    }
}
```

Example 2: Runtime Polymorphism (Method Overriding)

```
// Base class
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Derived class
```

```
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

// Derived class
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.sound(); // Calls Dog's sound method
        myCat.sound(); // Calls Cat's sound method
    }
}
```

Real-World Examples

1. **Payment System:**
    - **Superclass:** Payment

    - **Subclasses:** CreditCard, PayPal, BankTransfer
    - **Common Method:** `processPayment()`
    - Each subclass implements `processPayment()` differently based on payment type.
2. **Shape Drawing Application:**
    - **Superclass:** Shape
    - **Subclasses:** Circle, Square, Triangle
    - **Common Method:** `draw()`
    - Each shape draws differently but is treated as a general `Shape`.

Key Points

- **Method Overloading:** Same method name with different parameters within the same class.
- **Method Overriding:** Subclass changes the behavior of a method inherited from the parent class.
- **Dynamic Method Dispatch:** The method to be executed is determined at runtime.

Benefits of Polymorphism

- **Flexibility:** Write more generic code that works with objects of various classes.
- **Maintainability:** Simplifies code changes and additions.
- **Extensibility:** Easy to add new classes without modifying existing code.

Polymorphism enhances code flexibility and reusability, allowing systems to handle new requirements and changes with minimal impact.

3. Abstraction in Java

Definition: Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. It helps to reduce complexity and allows the user to interact with objects at a higher level without needing to understand the intricate details.

Types of Abstraction

1. **Abstract Classes:**
    - Classes that cannot be instantiated on their own and are meant to be extended by other classes.
    - Can contain abstract methods (methods without a body) and concrete methods (methods with an implementation).
2. **Interfaces:**
    - Abstract data types that specify a set of methods that implementing classes must provide.
    - Can only contain method signatures (abstract methods) and default methods (with an implementation, from Java 8 onwards).

Java Example: Abstract Classes

Abstract Class with Abstract Methods

```
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void sound();

    // Concrete method
    public void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.sound(); // Outputs: Dog barks
        myCat.sound(); // Outputs: Cat meows
        myDog.eat();   // Outputs: This animal eats food.
    }
}
```

Java Example: Interfaces

Interface Definition and Implementation

```
interface Animal {
    void sound(); // Abstract method
    void eat();   // Abstract method
}

class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }

    @Override
    public void eat() {
        System.out.println("Dog eats");
    }
}
```

```
class Cat implements Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }

    @Override
    public void eat() {
        System.out.println("Cat eats");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.sound(); // Outputs: Dog barks
        myCat.sound(); // Outputs: Cat meows
        myDog.eat();   // Outputs: Dog eats
        myCat.eat();   // Outputs: Cat eats
    }
}
```

Real-World Examples

1. **Vehicle System:**
    - **Abstract Class:** `Vehicle`
        - **Abstract Methods:** `start()`, `stop()`
        - **Concrete Method:** `fuelUp()`

    - **Concrete Classes:** `Car`, `Bike` that implement the `start()` and `stop()` methods.
    - Users interact with `Vehicle` to start or stop without needing to know how these actions are implemented.
2. **Remote Control:**
    - **Interface:** `RemoteControl`
        - **Methods:** `powerOn()`, `powerOff()`
    - **Concrete Implementations:** `TelevisionRemote`, `AirConditionerRemote`
    - Both remotes follow the same interface but have different internal mechanisms for `powerOn()` and `powerOff()`.

Key Points

- **Abstract Classes:** Allow partial implementation and can provide default behavior. They are suitable when multiple classes share common behavior but also have their own specific implementations.
- **Interfaces:** Provide a way to achieve abstraction and multiple inheritance. They define a contract that implementing classes must follow.

Benefits of Abstraction

- **Simplifies Code:** Hides complex implementation details, making the code easier to understand and use.
- **Increases Flexibility:** Allows changes to the implementation without affecting the code that uses the abstraction.
- **Promotes Reusability:** Encourages the reuse of code through abstract classes and interfaces.

Abstraction helps to manage complexity in software development by focusing on high-level functionalities while hiding the underlying implementation details.

Why OOP?

- **Modularity:** Code is organized into distinct classes.
- **Reusability:** Inherited classes can reuse existing code.
- **Scalability:** Easier to manage and extend code.
- **Maintainability:** Encapsulation makes it easier to change code without affecting other parts.

These concepts help in creating structured and efficient programs, making OOP a powerful paradigm for software development.

Assignment: Library Management System

Objective: Create a simplified library management system using object-oriented programming principles. This will involve designing classes and interfaces to handle books, users, and the library system itself.

Requirements:

1. **Abstract Class: `LibraryItem`**
    - **Abstract Methods:**
        - `String getItemDetails()`: Returns details of the item.
    - **Concrete Methods:**
        - `boolean isAvailable()`: Returns true if the item is available, otherwise false.
2. **Concrete Classes: `Book`, `Magazine`**
    - **Inheritance:** Both classes should inherit from `LibraryItem`.
    - **Attributes for `Book`:**
        - `String title`
        - `String author`
        - `boolean available`
    - **Attributes for `Magazine`:**
        - `String title`
        - `int issueNumber`
        - `boolean available`
    - **Methods:**
        - Implement `getItemDetails()` to return the specific details for books and magazines.
        - Implement other methods to manage availability (e.g., `borrow()`, `returnItem()`).
3. **Interface: `Borrowable`**
    - **Methods:**
        - `boolean borrow()`: Allows borrowing of the item if available.
        - `void returnItem()`: Returns the item and updates its status.
4. **Concrete Classes: `Book`, `Magazine`**
    - **Implement `Borrowable`:**
        - Implement `borrow()` and `returnItem()` methods to handle borrowing and returning of items.
5. **Class: `Library`**
    - **Attributes:**
        - `List<LibraryItem>` items
    - **Methods:**
        - `void addItem(LibraryItem item)`: Adds an item to the library.
        - `void removeItem(LibraryItem item)`: Removes an item from the library.
        - `LibraryItem searchItem(String title)`: Searches for an item by title and returns it if found.

6. **Class: `Main`**
    - **Create a library system and perform the following operations:**
        - Add several books and magazines to the library.
        - Display details of all items.
        - Borrow and return some items.
        - Search for an item by title.

Steps for Submission

1. Create the Java files as described.
2. Implement the classes and interfaces.
3. Test the functionality by running `Main.java` and verifying the output.

This assignment will help you practice abstraction, encapsulation, inheritance, and polymorphism while building a practical application.